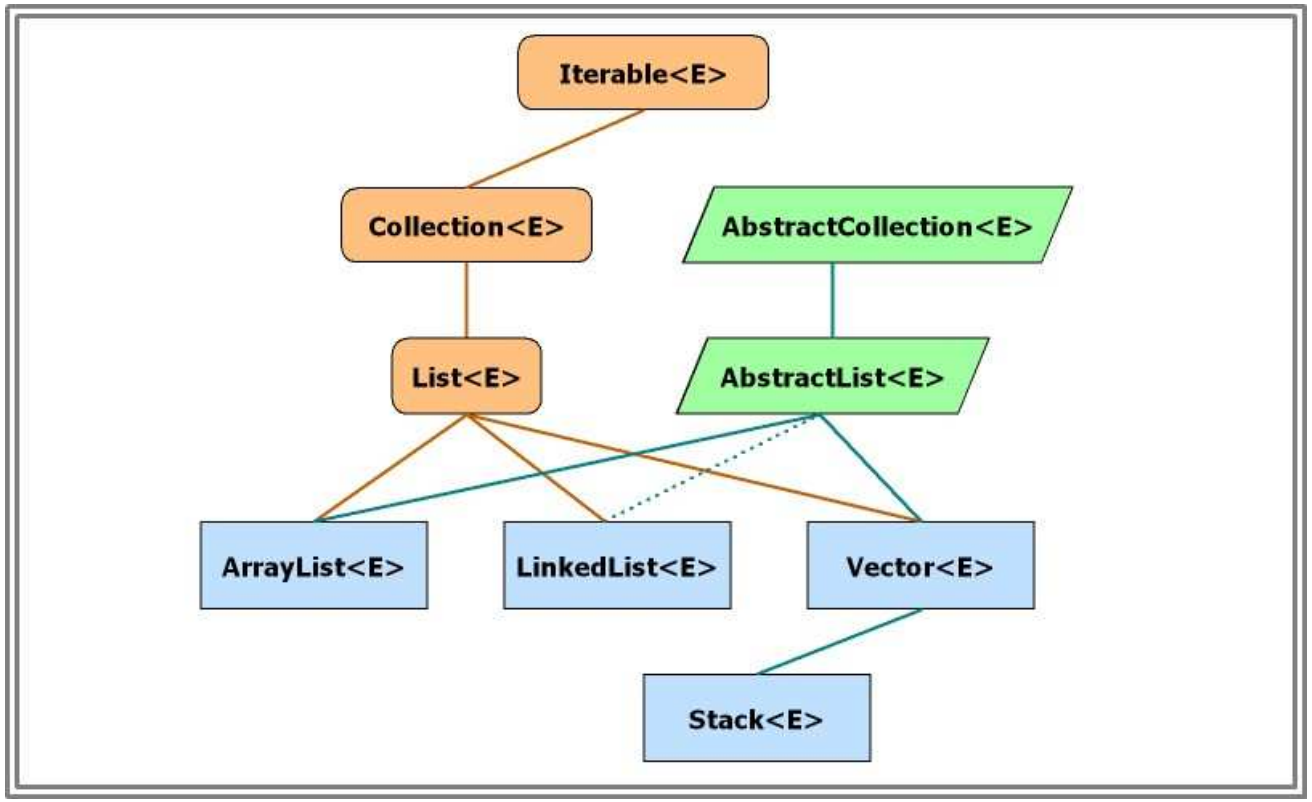
INTERFACES DE JAVA

O JCF é uma arquitectura que se baseia em três entidades fundamentais:

a) Interfaces;
b) Classes abstractas;
c) Classes Concretas – Implementações;



Interfaces são especificações sintácticas (algumas genéricas ie. Parametrizadas) de constantes e métodos que, em geral, correspondem a propriedades que qualquer classe que as pretenda ter/satisfazer pode implementar, desde que forneça implementações para todos os métodos especificados na interface.

```
public interface Colorivel {
    public abstract int getCor();
    public abstract void setCor(int cor);
}

public interface Datavel {
    void setData(GregorianCalendar data);
    GregorianCalendar getData();
}
```

Algumas Interfaces de Java nem sequer definem comportamento, são apenas etiquetas para o compilador/interpretador ("marker interfaces"):

```
public interface Serializable { }
```
/** As classes que declararem implementar esta interface podem ser gravadas em ficheiros de objectos, `ObjecStreams` */

```
public interface Cloneable { }
```
/** Classes que usam o método clone() de Object */ Nota: Não usamos !!

Interfaces são Tipos, tal como as classes, sendo compatíveis com o tipo das classes que as implementam, cf.

```
List<String> nomes = new ArrayList<String>();
                   = new LinkedList<Ponto2D>();
```

Interfaces são especificações implementadas pelas classes.

```
public class PontoComCor implements Colorivel {
    // variáveis de instância
    private int x, y;
    // variáveis para implementar Colorivel
    private int cor = 0;
    // métodos de instância
    ……..
    // implementação de Colorivel
    public int getCor() { return cor; };
    public void setCor(int ncor) { cor = ncor; };
}
```

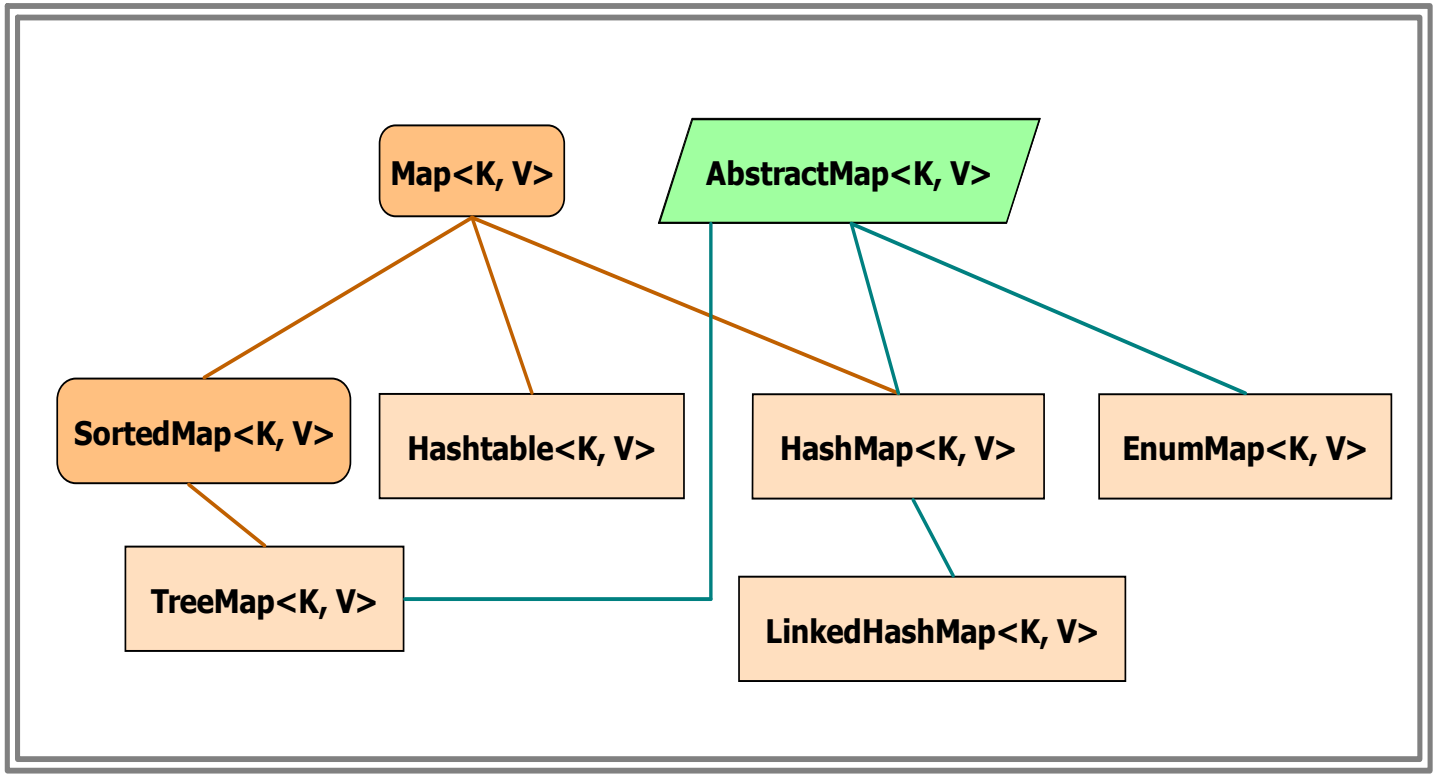
Assim:

```
PontoComCor pc = new PontoComCor();
Colorivel col = new PontoComCor(10.0, 5.0);

x = pc.getX(); cor = pc.getCor(); cor = col.getCor(); col.setCor(99);
ERRADAS:col.getX(); col.somaPonto(1.0, 5.0);    /** cf. as suas API */
```

Uma Classe pode implementar várias interfaces:

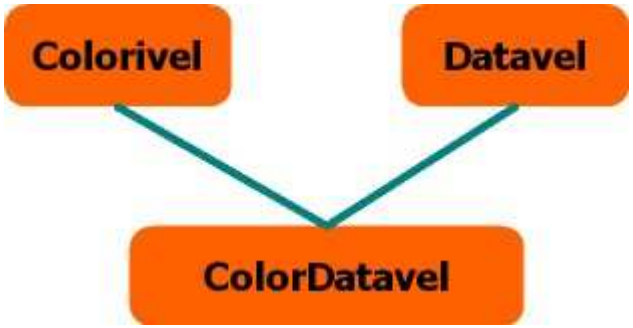
```
public class PontoComCorDatavel implements Colorivel, Datavel {
    // variáveis de instância
    private int x, y;
    // variáveis para implementar Colorível
    private int cor = 0;
    // variáveis para implementar Datavel
    private GregorianCalendar data = null;

    // métodos de instância
    ……..
    // implementação de Colorivel
    public int getCor() { return cor; };
    public void setCor(int ncor) { cor = ncor; };

    // implementação de Datavel
    public void setData(GregorianCalendar data) {
        this.data = (GregorianCalendar) data.clone();
    }
    public GregorianCalendar getData() {
       return (GregorianCalendar) data.clone();
    }
}
```

As Interfaces de Java podem fazer parte de uma hierarquia de interfaces. Mas podem ser interfaces livres, desligadas de todas as outras. Por vezes dá mesmo jeito colocar certas interfaces numa hierarquia múltipla. Por exemplo, em JCF:



Assim, a classe `TreeMap<K,V>` implementa a interface `SortedMap<K,V>` o que significa que por herança, implementa a interface `Map<K,V>` mais os métodos adicionais especificados em `SortedMap<K,V>`.

No nosso exemplo anterior, poderíamos ter:



E então a classe `PontoComCorDatavel` Poderia ser declarada apenas como:

```
public class PontoComCorDatavel implements ColorDatavel {
```

Algumas interfaces de Java são muito importantes. Por exemplo a interface `Comparable<T>` definida como:

```
public interface Comparable<T> {
  int compareTo(T obj);
}
```

Qualquer classe que pretenda ter uma **ordem natural** (>, = <) definida para as suas instâncias **deve implementar esta interface**. A classe `String` e as classes "Wrapper" implementam esta interface `Comparable<T>` e por isso são automaticamente ordenadas em `TreeSet<E>` e `TreeMap<K,V>`.

Porém, o que fazer se se pretender ordenar objectos fora da sua ordem natural ou se se pretender ordenar objectos que não implementam `Comparable<T>`?

Uso da interface `Comparator<E>` em `TreeSet<E>` (e `TreeMap<K,V>` para `K`)

As colecções implementadas usando árvores possuem um construtor que permite que se defina um algoritmo de comparação a usar na ordenação dos objectos. No caso, por exemplo, de `TreeSet<E>`,

```
TreeSet<E>(Comparator<E>)
```

O parâmetro `Comparator<E>` é uma interface Parametrizada de Java, que está definida como:

```
public interface Comparator<T> {
   int compare(T obj1, T obj2); // 0, 1 ou -1
}
```

Assim, se, por exemplo, pretendermos criar um `TreeSet<Ponto2D>` de forma a que os pontos estejam ordenados, temos que criar uma classe que implemente a interface `Comparator<Ponto2D>`, ou seja, implemente o método

```
int compare(Ponto2D obj1, Ponto2D obj2)
```

Vejamos como poderia ser tal classe (1 de muitas hipóteses):

```
import java.util.Comparator;
public class PontoComparator implements Comparator<Ponto2D> {
   public int compare(Ponto2D p1, Ponto2D p2) {
     if(p1.getX() < p2.getX()) return -1;
     if(p1.getX() > p2.getX()) return 1;
     if(p1.getY() < p2.getY()) return -1;
     if(p1.getY() > p2.getY()) return 1; else return 0;
   }
}
```

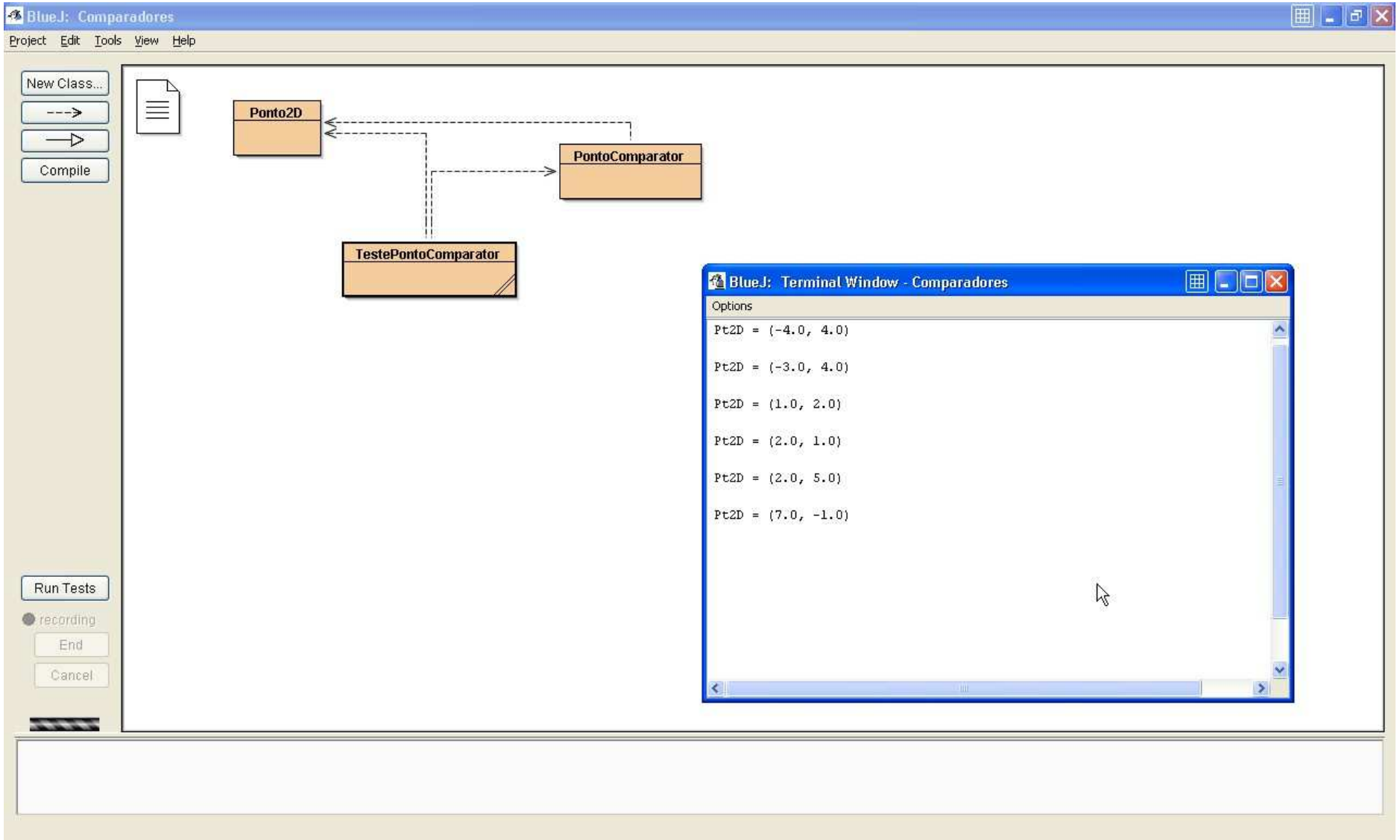
Temos pois uma classe que implementa a interface `Comparator<Ponto2D>`. Vamos agora testá-la, criando um `TreeSet<Ponto2D>` usando o construtor na forma

```
new TreeSet<Ponto2D>(new PontoComparator())
```

e escrevendo um programa de teste que vai verificar se os elementos do conjunto são de facto enumerados conforme o algoritmo de ordenação definido.

```
/**
 * TestePontoComparator
 *
 * @author F. Mário Martins
 * @version 1/2006
 */
import static java.lang.System.out;
import java.util.*;
public class TestePontoComparator {

    public static void main(String args[]) {
        TreeSet<Ponto2D> plano =
                    new TreeSet<Ponto2D>(new PontoComparator());
        plano.add( new Ponto2D(7.0, -1.0) );
        plano.add( new Ponto2D(1.0, 2.0) );
        plano.add( new Ponto2D(-3.0, 4.0) );
        plano.add( new Ponto2D(2.0, 5.0) );
        plano.add( new Ponto2D(-4.0, 4.0) );
        plano.add( new Ponto2D(2.0, 1.0) );

        for(Ponto2D p : plano) out.println(p.toString());
    }

}
```

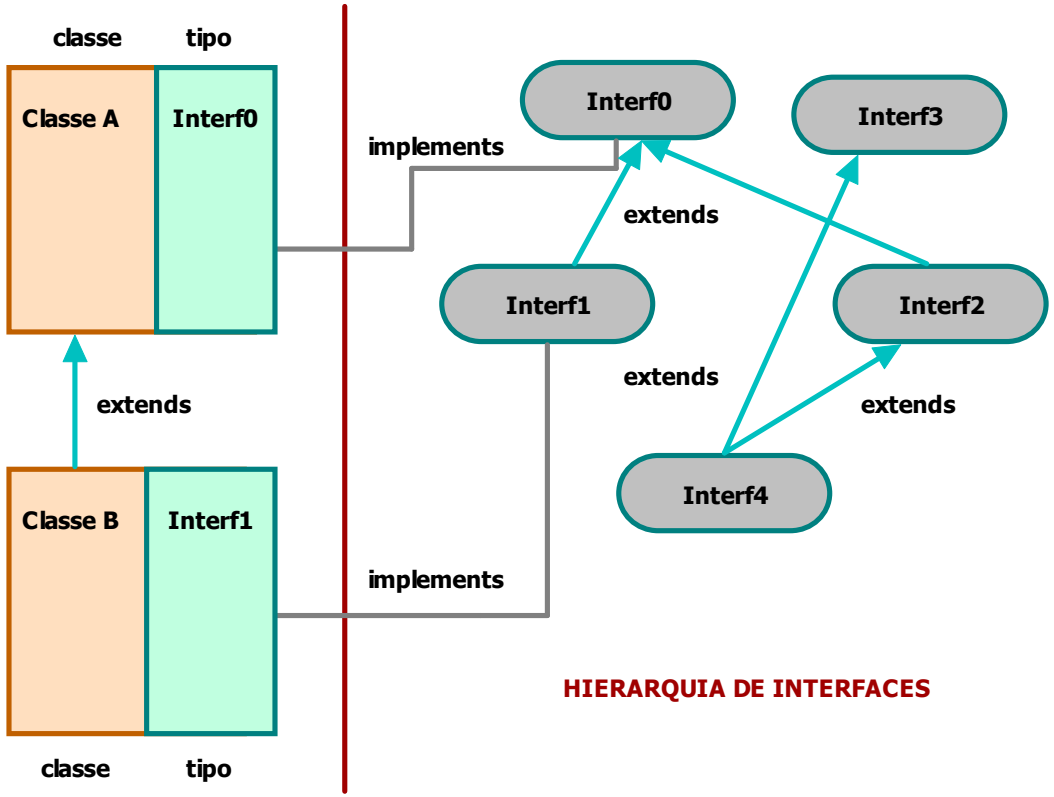
RESULTADOS:


BlueJ: Comparadores
Project Edit Tools View Help
New Class...
Compile
Run Tests
recording
End
Cancel
Ponto2D
PontoComparator
TestePontoComparator
BlueJ: Terminal Window - Comparadores
Options
Pt2D = (-4.0, 4.0)
Pt2D = (-3.0, 4.0)
Pt2D = (1.0, 2.0)
Pt2D = (2.0, 1.0)
Pt2D = (2.0, 5.0)
Pt2D = (7.0, -1.0)

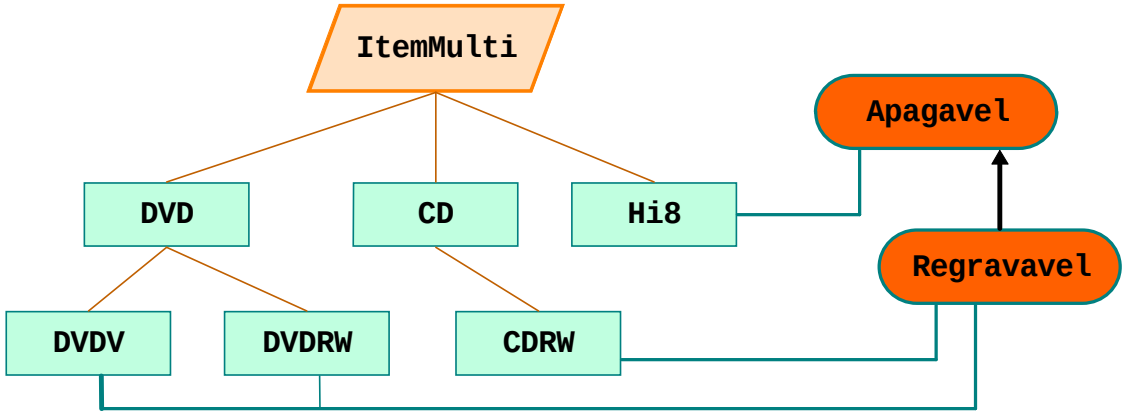
Relembrando a classe `Plano`, vamos redefini-la agora como,

```
public class Plano implements Serializable {
    private TreeSet<Ponto2D> pontos;

    // Construtor por omissão: usa o comparador PontoComparator
    public Plano() { pontos = new TreeSet<Ponto2D>( new PontoComparator() ); }
    // Recebe um conjunto de pontos e um algoritmo de ordenação e copia os
    // pontos para a variável de instância, ordenando-os por tal ordem
    public Plano(Set<Ponto2D> pts, Comparator<Ponto2D> comp) {
        pontos = new TreeSet<Ponto2D>( comp );
        for(Ponto2D p : pts) pontos.add(p.clone());
    }

    // Métodos de instância

    ....

    // Método que devolve uma cópia do TreeSet<Ponto2D>
    // ordenado pelo algoritmo de ordenação passado como parâmetro
    public TreeSet<Ponto2D> ordenaPor(Comparator<Ponto2D> comp) {
       TreeSet<Ponto2D> aux = new TreeSet<Ponto2D>(comp);
            for(Ponto2D p : pontos) aux.add(p.clone());
       return aux;
    }
}
```

Classes e Interfaces JAVA pertencem a duas hierarquias distintas. A hierarquia de classes é de tipo simples enquanto que a de classes é uma hierarquia múltipla (1 interface pode herdar de mais do que uma).



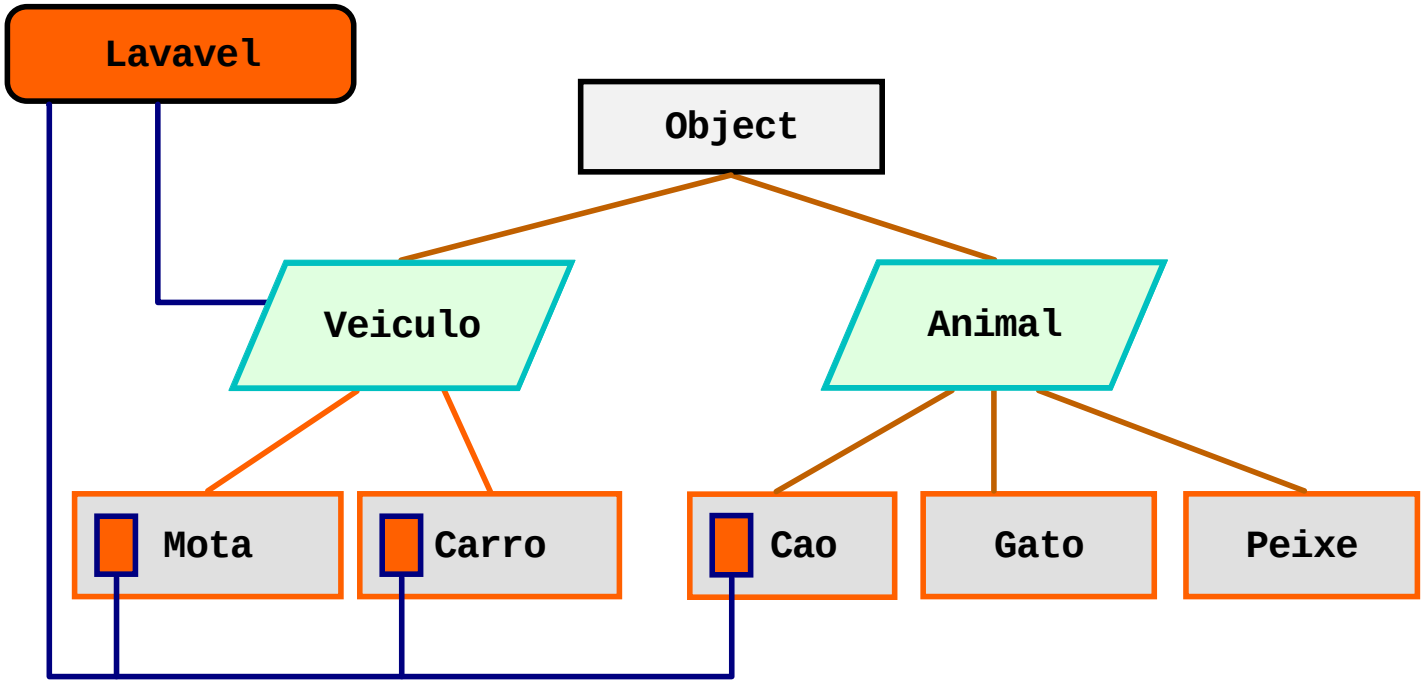
EXEMPLOS:




```
public interface Apagavel {
  public abstract void apaga();
}

public interface Regravavel extends Apagavel {
  public abstract void grava(byte[] cont);
}
```

INTERFACES PODEM SER IMPLEMENTADAS POR QUAISQUER CLASSES EM QUALQUER PONTO DA HIERARQUIA

```
public interface Lavavel {
  public void aLavar(int tempo);
}
```



INTERFACES NÃO INTERFEREM NA HIERARQUIA DE CLASSES

INTERFACES VERSUS CLASSES ABSTRACTAS

A TABELA APRESENTA MAIS ALGUMAS DIFERENÇAS ENTRE A UTILIZAÇÃO DE INTERFACES E DE CLASSES ABSTRACTAS, RELATIVAMENTE A PROPRIEDADES DIVERSAS, NÃO TANTO CONCEPTUAIS MAS MAIS RELACIONADAS COM A FUNCIONALIDADE E RESTRIÇÕES NA UTILIZAÇÃO.

Propriedades	Interfaces	Classes Abstractas
Herança Múltipla	Uma classe pode implementar várias interfaces	Uma classe só pode "estender" uma classe abstracta
Implementação	Não aceita nenhum código	O código que se pretender, cf. as necessidades da concepção
Constantes	Apenas `static final`. Podem ser usadas pelas classes implementadoras sem qualificadores	De classe ou de instância e código de inicialização se necessário
Relação *is-a*	Em geral não descrevem a identidade principal de uma classe, mas capacidades periféricas aplicáveis a outras classes	Descreve uma identidade básica de todos os seus descendentes
Adicionar funcionalidade	Não deve ser feito pois implicaria refazer todas as implementações da interface. Usar antes subinterfaces	Sem problema para as subclasses desde que o método seja codificado na classe abstracta
Homogeneidade	Classes que as implementam são homogéneas na API	Subclasses homogéneas em API e que partilham algum código, por exemplo, classes de dados
Velocidade (não é relevante)	Requer extra indirecção na procura	Procura directa do método

Interfaces *versus* classes abstractas